International Computer Science Competition Solutions for Qualification Round

Ashish Padhy
NIT Rourkela
ashishpadhy1729@gmail.com

June 12, 2025

1 Problem A: Neural Network Components

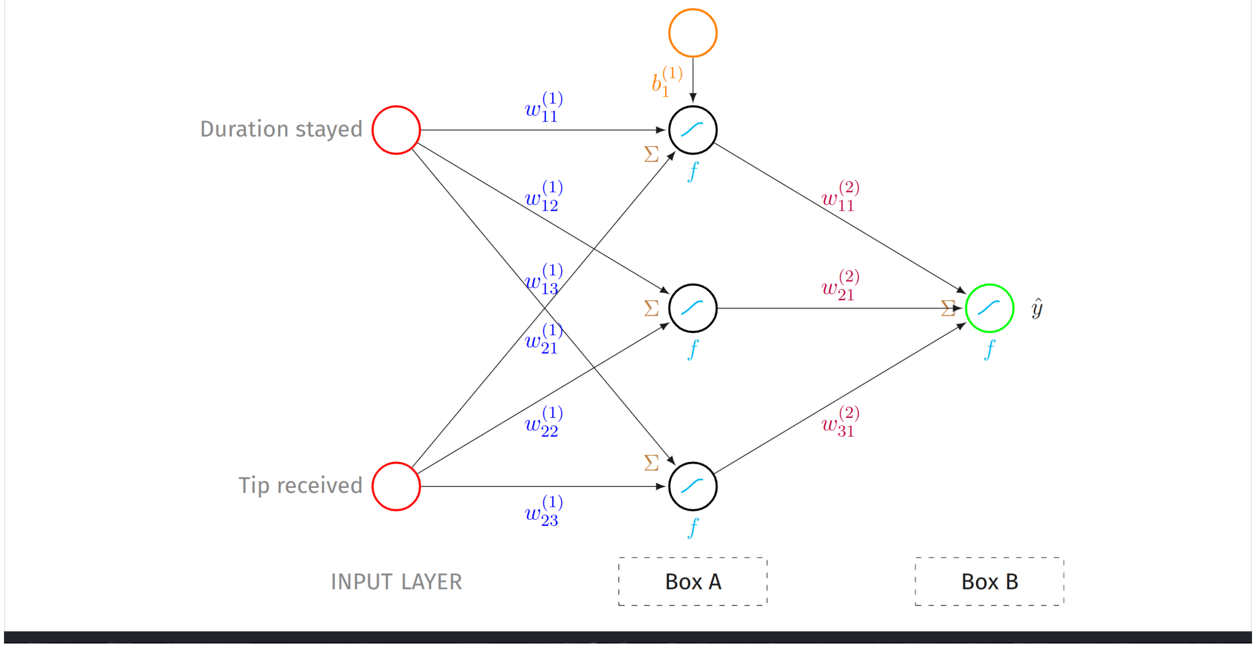


Figure 1: The neural network architecture given in Problem A

Below is the description of the components asked in the question with reference to the neural network diagram 1.

$w_{21}^{(1)}$: Weight from **input node 2** (*Tip received*) to the first neuron in the hidden layer (Layer 1).

Σ: Represents the **weighted sum** operation (linear combination of inputs and bias), i.e. $\Sigma = \Sigma_{i=1}^{n} w_{ij}^{(l)} x_i^{(l)} + b_j^{(l)}$.

f: Denotes the **activation function** applied after the weighted sum. Example: Sigmoid function, ReLU, etc.

Red Circle: Input nodes — specifically, *Duration stayed* and *Tip received.*

Orange Circle: Bias node — a constant input contributing to the neuron's activation.

Green Circle: Output node — responsible for computing the predicted output $\hat{y}$.

Box A: This represents the **hidden layer** — neurons that compute intermediate representations using inputs.

Box B: This represents the **output layer** — final layer producing the prediction.

$\hat{y}$: The **predicted output** of the neural network (e.g., customer satisfaction).

Summary

In this feedforward neural network, each input is passed through weighted connections, combined using a weighted sum (Σ), activated using a function f, and finally produces an output $\hat{y}$. Biases and intermediate hidden layers (Box A) help in capturing complex patterns. The structure represents a classic multi-layer perceptron (MLP). Finally, the output layer (Box B) produces the predicted output $\hat{y}$. This can be used to calculate the loss by comparing the predicted output with the actual output.

2 Problem B: Cake Calculator

Implementation

Below is the implementation of the **Cake Calculator** in C++:

```cpp
#include <iostream>
#include <vector>
#include <stdexcept>
#include <cassert>

std::vector<int> cake_calculator(int flour, int sugar) {
    if (flour <= 0 || sugar <= 0) {
        throw std::invalid_argument("Flour and sugar must be greater than 0");    // Check if both flour and sugar are positive
    }

    // Variables for the amount of flour and sugar per cake
    int flour_per_cake = 100;
    int sugar_per_cake = 50;

    // Number of cakes that can be made is the minimum of the number of cakes that can be made from flour and sugar
    int num_cakes = std::min(flour / flour_per_cake, sugar / sugar_per_cake);

    // Calculate the remaining flour and sugar after making the cakes
    int remaining_flour = flour - num_cakes * flour_per_cake;
    int remaining_sugar = sugar - num_cakes * sugar_per_cake;

    return std::vector<int>{num_cakes, remaining_flour, remaining_sugar};
}
```

Explanation

The function `cake_calculator` calculates how many cakes can be made with the given amounts of flour and sugar, and also returns the remaining amounts of each ingredient after making those cakes. The function works as follows:

1. It first checks if the inputs for flour and sugar are positive (throws `invalid_argument`).
2. It calculates the maximum number of cakes that can be made by taking the minimum of the number of cakes that can be made from the available flour and sugar. This is done using the formula:

$$\text{num\_cakes} = \min\left(\frac{\text{flour}}{\text{flour\_per\_cake}}, \frac{\text{sugar}}{\text{sugar\_per\_cake}}\right) \tag{1}$$

   where `flour_per_cake` is 100 and `sugar_per_cake` is 50.
3. Finally, it calculates the remaining amounts of flour and sugar and returns them in a vector.

3 Problem C: The School Messaging App

Given, for a set of symbols with probabilities $p_1, p_2, \dots, p_n$, the entropy H is defined as:

$$H = -\sum_{i=1}^{n} p_i \log_2(p_i) \tag{2}$$

Question 1: Standard text encodings use the same number of bits for each character. Why would using different length codes for characters with different probabilities help you transmit more messages within your data limit? Give an example.

When we use fixed length codes for characters, each character occupies the same number of bits regardless of its frequency of occurrence. This leads to wasted space as characters that occur more frequently could be represented with fewer bits decreasing the average code length. Variable length codes, such as **Huffman coding or Fano coding**, allow us to assign shorter codes to more frequent characters and longer codes to less frequent characters. This results in a more efficient encoding scheme. The same thing is represented mathematically by the entropy formula (2) because if we take bits equal to the $log_2(1/p_i)$ for each character, we take higher bits for less frequent characters.

Example: Consider a character set with the following probabilities:

$$p_i : \quad a : 0.5 \qquad b : 0.25 \qquad c : 0.125 \qquad d : 0.125$$

Using **fixed-length encoding**, we might use 2 bits for each character:

$$a : 00 \qquad b : 01 \qquad c : 10 \qquad d : 11$$

Using **variable-length encoding**, we could assign:

$$a : 0 \qquad b : 10 \qquad c : 110 \qquad d : 111$$

$$\text{Average Length} = 0.5 \times 1 + 0.25 \times 2 + 0.125 \times 3 + 0.125 \times 3 = 1.625 \text{ bits}$$

This is significantly less than fixed-length encoding, allowing us to transmit more messages within the same data limit. The percentage decrease is $\frac{2-1.625}{2} \times 100\% = \underline{\mathbf{18.75\%}}$.

Question 2: Calculate the entropy for the 12-character set above. What does this value represent in terms of optimal encoding?

The entropy H for the character set can be calculated using equation (2):

$$\begin{aligned}
H = -\big(&0.20\log_2(0.20) + 0.15\log_2(0.15) + 0.12\log_2(0.12) + 0.10\log_2(0.10)\\
&+ 0.08\log_2(0.08) + 0.06\log_2(0.06) + 0.05\log_2(0.05) + 0.05\log_2(0.05)\\
&+ 0.04\log_2(0.04) + 0.03\log_2(0.03) + 0.02\log_2(0.02) + 0.10\log_2(0.10)\big)\\
= -\big(&0.20\times(-2.3219) + 0.15\times(-2.7369) + 0.12\times(-3.0589) + 0.10\times(-3.3219)\\
&+ 0.08\times(-3.6439) + 0.06\times(-4.0589) + 0.05\times(-4.3219) + 0.05\times(-4.3219)\\
&+ 0.04\times(-4.6439) + 0.03\times(-5.0589) + 0.02\times(-5.6439) + 0.10\times(-3.3219)\big)\\
= \ &0.4644 + 0.4105 + 0.3671 + 0.3322 + 0.2915 + 0.2435+\\
&0.2161 + 0.2161 + 0.1858 + 0.1518 + 0.1129 + 0.3322\\
= \ &\underline{\textbf{3.3239 bits}} \qquad (3)
\end{aligned}$$

This value represents the **theoretical minimum average number of bits per character** needed to encode the set of characters optimally.

Question 3: Calculate the average code length of your Fano code and compare it to the theoretical entropy limit. How efficient is your Fano code?

The average code length of the Fano code shown in the problem can be calculated as follows:

$$\begin{aligned}
\text{Average Length} &= 0.20\times 3 + 0.15\times 3 + 0.12\times 3 + 0.10\times 4 + 0.08\times 4 + 0.06\times 4+\\
&\quad 0.05\times 3 + 0.05\times 4 + 0.04\times 4 + 0.03\times 4 + 0.02\times 4 + 0.10\times 4\\
&= 0.60 + 0.45 + 0.36 + 0.40 + 0.32 + 0.24 + 0.15 + 0.20 + 0.16 + 0.12 + 0.08 + 0.40\\
&= \underline{\textbf{3.48 bits}} \qquad (4)
\end{aligned}$$

Given the entropy $H = 3.32$ bits (from Question 2, Eqn.(3)), the Fano code has an average length of 3.48 bits.

This means the efficiency of the Fano code is:

$$\textbf{Efficiency} = \frac{H}{\text{Average Length}} = \frac{3.32}{3.48} \approx \underline{\textbf{0.954 (or 95.4\%)}}$$

Thus, the Fano code is about **95.4% efficient**, which is close to optimal but slightly longer than the theoretical minimum entropy.

4 Problem D: Word Search Puzzle

4.1 Implementation

The following C++ code implements the solution to the **Word Search Puzzle** problem:

```cpp
#include <iostream>
#include <vector>
#include <string>
#include <cstdlib>
#include <ctime>
#include <algorithm>
#include <stdexcept>
#include <sstream>
#include <unordered_map>

// Puzzle grid size constants
const int ROW_SIZE = 10;
const int COL_SIZE = 10;

// Direction structure to represent movement in the grid, dr for delta row,
  dc for delta column, label for debugging
struct Direction {
    int dr, dc;
    std::string label;
};

/**
* Structure to represent a word placement in the grid.
* It contains the word itself, its starting row and column, and the
  direction of placement.
*/
struct WordPlacement {
    std::string word;
    int row, col;
    Direction direction;
};

// Valid directions for word placement
const std::vector<Direction> directions = {
    {0, 1, "→"},     // right
    {1, 0, "↓"},     // down
    {1, 1, "↘"},     // diagonal down-right
    {-1, 1, "↗"}     // diagonal up-right
};

/**
* Normalize the words by converting them to uppercase.
* This is necessary to ensure case-insensitive matching in the grid.
* @param words The vector of words to normalize.
*/
void normalizeWords(std::vector<std::string>& words) {
    for (std::string& word : words) {
        std::transform(word.begin(), word.end(), word.begin(), ::toupper);
    }
}

/**
* Check if a word can be placed in the grid at the specified position and direction.
* The word should follow the Satisfiability Condition.
* @param grid The 2D vector representing the word search grid.
* @param word The word to check for placement.
* @param row The starting row index for placement.
* @param col The starting column index for placement.
* @param dr The delta row for direction (1 for down, 0 for right, etc.).
* @param dc The delta column for direction (1 for right, 0 for down, etc.).
* @return True if the word can be placed, false otherwise.
*/
bool canPlaceWord(const std::vector<std::vector<char>>& grid, const std::string& word,
                  int row, int col, int dr, int dc) {
    int r = row, c = col;
    for (char ch : word) {
        if (r < 0 || r >= ROW_SIZE || c < 0 || c >= COL_SIZE ||
            (grid[r][c] != '.' && grid[r][c] != ch)) {
            return false;
        }
        r += dr;
        c += dc;
    }
    return true;
}

/**
* Place a word in the grid at the specified position and direction.
* This function modifies the grid in place and stores the positions of the letters for undoing if necessary.
* @param grid The 2D vector representing the word search grid.
* @param word The word to place in the grid.
* @param row The starting row index for placement.
* @param col The starting column index for placement.
* @param dr The delta row for direction (1 for down, 0 for right, etc.).
* @param dc The delta column for direction (1 for right, 0 for down, etc.).
* @param positions A vector to store the positions of the letters placed.
*/
void placeWord(std::vector<std::vector<char>>& grid,
               const std::string& word, int row, int col, int dr, int dc,
               std::vector<std::pair<int, int>>& positions) {
    int r = row, c = col;
    for (char ch : word) {
        if (grid[r][c] == '.') {
            grid[r][c] = ch;
            positions.push_back({r, c});  // Store only new placements for undo
        }
        r += dr;
        c += dc;
```

```cpp
        }
    }

    /**
    * Undo the placement of a word by resetting the grid positions to '.'.
    * This is used to backtrack when a word placement fails.
    * @param grid The 2D vector representing the word search grid.
    * @param positions The list of positions that were occupied by the word.
    */
    void undoPlacement(std::vector<std::vector<char>>& grid,
                      const std::vector<std::pair<int, int>>& positions) {
        for (const auto& [r, c] : positions) {
            grid[r][c] = '.';
        }
    }

    /**
    * Recursive backtracking function to place all words in the grid
    * @param grid The 2D vector representing the word search grid.
    * @param words The list of words to place in the grid.
    * @param index The current index in the words list.
    * @param wordPlacements A map to store the placements of words.
    * @return True if all words are placed successfully, false otherwise.
    */
    bool placeAllWordsBacktracking(std::vector<std::vector<char>>& grid,
                                  const std::vector<std::string>& words,
                                  int index,
                                  std::unordered_map<std::string, WordPlacement
     >& wordPlacements) {

        if (index == words.size()) return true;  // All words placed

        const std::string& word = words[index];

        // Try to place the current word in all possible positions and
     directions
        for (const Direction& dir : directions) {
            // Calculate the maximum row and column indices based on the word
     length and direction
            int maxRow = (dir.dr == -1) ? ROW_SIZE - 1 : ROW_SIZE - dir.dr * (
     int)word.length();
            int maxCol = COL_SIZE - dir.dc * (int)word.length();

            for (int row = 0; row <= maxRow; ++row) {
                for (int col = 0; col <= maxCol; ++col) {
                    if (canPlaceWord(grid, word, row, col, dir.dr, dir.dc)) {
                        std::vector<std::pair<int, int>> placedPositions;
                        placeWord(grid, word, row, col, dir.dr, dir.dc,
     placedPositions);
                        wordPlacements[word] = {word, row, col, dir};

                        if (placeAllWordsBacktracking(grid, words, index + 1,
     wordPlacements)) {
                            return true;
                        }

                        // Restore the grid to its previous state if placement failed
                        undoPlacement(grid, placedPositions);
                    }
                }
            }
        }

    return false; // No valid placement for this word
}


/**
* Fill the grid with random letters in empty cells (denoted by '.').
* This function is called after all words have been placed.
* @param grid The 2D vector representing the word search grid.
*/
void fillGridWithRandomLetters(std::vector<std::vector<char>>& grid) {
    for (auto& row : grid) {
        for (char& cell : row) {
            if (cell == '.') {
                cell = 'A' + std::rand() % 26;
            }
        }
    }
}

/**
* Generate a 10x10 word search puzzle containing the given words.
*
* @param words A vector of strings to include in the puzzle.
* @return A 2D vector of chars representing the word search puzzle.
*/

std::vector<std::vector<char>> generateWordSearch(const std::vector<std::string>& inputWords) {
    std::srand(std::time(nullptr));
    std::vector<std::vector<char>> grid(ROW_SIZE, std::vector<char>(COL_SIZE, '.'));

    std::vector<std::string> words = inputWords;
    std::unordered_map<std::string, WordPlacement> wordPlacements;
    normalizeWords(words);

    if (!placeAllWordsBacktracking(grid, words, 0, wordPlacements)) {
        throw std::runtime_error("Failed to place all words with backtracking. The word list may be too long or incompatible.");
    }

    fillGridWithRandomLetters(grid);

    for (const auto& [word, placement] : wordPlacements) {
        std::cout << "Placed word: " << word << " at (" << placement.row << ", " << placement.col
                  << ") in direction " << placement.direction.label << std::endl;
    }
    return grid;
}
```

4.2 Explanation

The Word Search Puzzle generator creates a 10x10 grid in which a given list of words is hidden in various directions. The algorithm ensures that the words do not conflict and that the remaining cells are filled with random letters to resemble a typical word search puzzle. The implementation uses recursive backtracking to attempt to place all words in the grid.

- **Grid Initialization:** A 10x10 grid of characters is initialized with the character `'.'` to represent empty cells.
- **Normalization:** All input words are converted to uppercase to maintain uniformity and to allow for consistent matching.
- **Valid Directions:** The words can be placed in four directions: right ($\rightarrow$), down ($\downarrow$), diagonal down-right ($\searrow$), and diagonal up-right ($\nearrow$). These directions are represented using the `Direction` struct.
- **Backtracking Algorithm:**
  - The recursive function `placeAllWordsBacktracking()` attempts to place each word one by one.
  - For each word, it iterates through all positions and directions and uses `canPlaceWord()` to verify if a placement is valid.
  - If valid, it uses `placeWord()` to write the word into the grid, and stores the affected cells.
  - If at any point a word cannot be placed, it undoes the previous placement using `undoPlacement()` and backtracks to try a different configuration.
- **Random Fill:** Once all words are successfully placed, any remaining empty cells ('.') are filled with random uppercase letters using `fillGridWithRandomLetters()`.
- **Word Placement Tracking:** For debugging and analysis, the placement of each word (its coordinates and direction) is printed to the console.
- **Failure Case:** If the algorithm fails to place all words (due to constraints like too many words or incompatible word lengths), it throws a `std::runtime_error` exception.

This approach ensures that all words are placed without conflict, and any failure leads to graceful backtracking. The use of randomized filling in empty cells ensures that the puzzle looks complete and prevents the user from visually guessing based on blank spaces.

4.3 Test Cases

The following command was used to compile and run the code:

```
>> g++ problem_d.cpp; ./a.out
```

Debug output (word placements with their coordinates and directions):

```
WARP, PRONG, GNARL, LAP, PERIL, LEND, DART, TAR, RANT, TEN, NODE, DEAL, PLAN,
    PANE, NEAT, STAG, GRANT, LEAD, DROP, PORT,TIDE, RIND, RUNG, GLEN, DENT,
    STAR, RIDE, GLOW, REAL

Placed word: REAL at (3, 2) in direction ↘
Placed word: RIDE at (9, 3) in direction ↗
Placed word: STAR at (9, 0) in direction →
Placed word: DENT at (8, 5) in direction →
Placed word: GLOW at (5, 9) in direction ↓
Placed word: PORT at (8, 0) in direction →
Placed word: DROP at (7, 5) in direction →
Placed word: LEAD at (6, 5) in direction →
Placed word: RIND at (7, 0) in direction →
Placed word: STAG at (5, 6) in direction →
Placed word: PANE at (5, 0) in direction →
Placed word: WARP at (0, 0) in direction →
Placed word: NODE at (4, 0) in direction →
Placed word: GLEN at (9, 6) in direction →
Placed word: DEAL at (4, 2) in direction →
Placed word: TIDE at (8, 3) in direction →
Placed word: LAP at (1, 4) in direction →
Placed word: GNARL at (1, 0) in direction →
Placed word: DART at (3, 0) in direction →
Placed word: TEN at (3, 7) in direction →
Placed word: GRANT at (6, 0) in direction →
Placed word: PERIL at (2, 0) in direction →
Placed word: LEND at (2, 4) in direction →
Placed word: NEAT at (5, 2) in direction →
Placed word: PRONG at (0, 3) in direction →
Placed word: TAR at (1, 7) in direction →
Placed word: RUNG at (9, 3) in direction →
Placed word: RANT at (3, 4) in direction →
Placed word: PLAN at (4, 6) in direction →
```

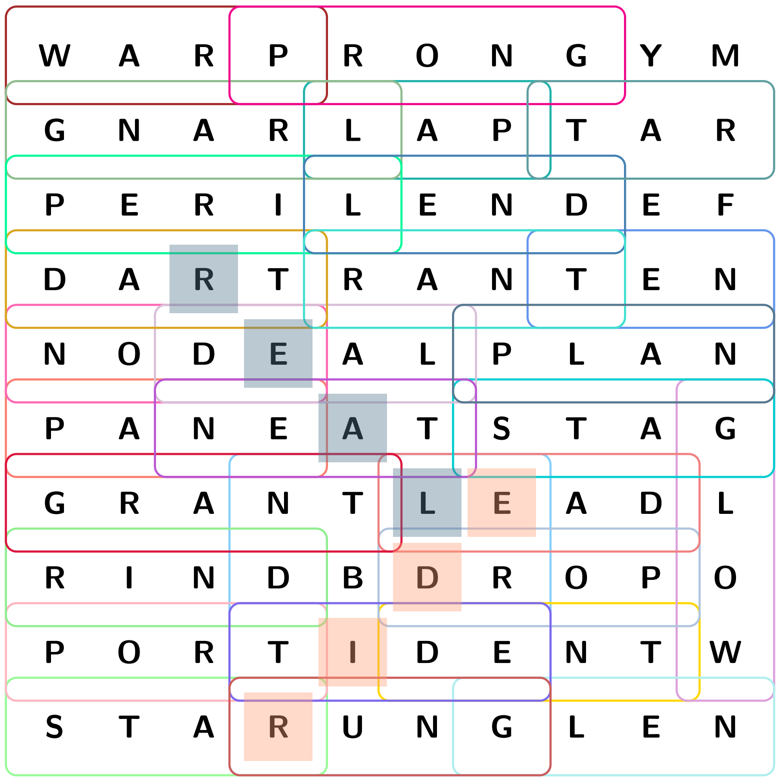


Figure 2: Word Search Puzzle Grid with rounded rectangles around words. **Note: Diagonal words could not be rounded with rectangles (REAL, RIDE) hence their cells are highlighted with a different color.**

5 Problem E: Functional Completeness of NAND

A NAND (NOT AND) gate can be represented as:

$$\text{NAND}(A, B) = A \uparrow B = \neg(A \wedge B) \tag{5}$$

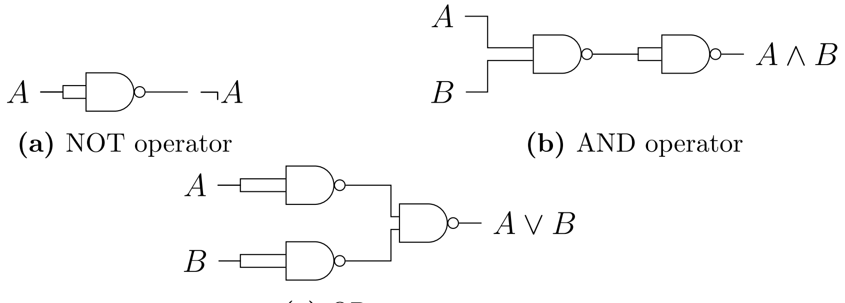
Hence, the truth table for NAND is:

A	B	NAND(A, B)
0	0	1
0	1	1
1	0	1
1	1	0

Definition 5.1. *A set of `logical connectives` or `Boolean operators` is said to be **functionally complete** if any boolean function (all possible combinations of truth values for a given number of variables) can be expressed using just those operators.*

The NAND gate can be used to construct the basic gates as shown below and in figure 3.

- **NOT** gate: $\text{NOT}(A) = \neg A = \neg(A \wedge A) = \text{NAND}(A, A)$
- **AND** gate: $A \wedge B = \neg(\neg A \wedge \neg B) = \neg(\text{NAND}(A, B)) = \text{NAND}(\text{NAND}(A, B), \text{NAND}(A, B))$ (From derivation of NOT gate)
- **OR** gate: $A \vee B = \neg(\neg A \wedge \neg B) = \text{NAND}(\neg A, \neg B) = \text{NAND}(\text{NAND}(A, A), \text{NAND}(B, B))$ (Using De Morgan's Law)



(a) NOT operator **(b)** AND operator

(c) OR operator

Figure 3: Basic logic gates constructed using only NAND Gates

Theorem 5.1. *Every Boolean function can be expressed in either Disjunctive Normal Form (DNF) or Conjunctive Normal Form (CNF). That is, for any Boolean function $f(x_1, x_2, \ldots, x_n)$, there exists an equivalent expression composed entirely of conjunctions (AND), disjunctions (OR), and negations (NOT) in one of the following forms:*

$$f(x_1, x_2, \ldots, x_n) = \bigvee_{i=1}^{m} \left(\bigwedge_{j=1}^{n_i} l_{ij} \right) \quad \textit{(DNF)} \qquad \textit{or} \qquad f(x_1, x_2, \ldots, x_n) = \bigwedge_{i=1}^{k} \left(\bigvee_{j=1}^{m_i} l_{ij} \right) \quad \textit{(CNF)} \tag{6}$$

where l_{ij} is either a variable x_j or its negation $\neg x_j$.

From Theorem 5.1, we can conclude that the set $\{\text{AND}, \text{OR}, \text{NOT}\}$ is functionally complete, as any Boolean function can be constructed using only these logical connectives by expressing it in DNF or CNF.

Furthermore, since each of the connectives AND, OR, and NOT can be implemented solely using NAND gates, it follows that:

The NAND gate is functionally complete, as it can be used to construct any Boolean function.